FRAIGs: Functionally Reduced AND-INV Graphs

Alan Mishchenko, Roland Jiang, Satrajit Chatterjee, Robert Brayton

Department of EECS, University of California, Berkeley

{alanmi, jiejiang, satrajit, brayton}@eecs.berkeley.edu

Abstract

AND-INV graphs (AIGs) are Boolean networks composed of two-input AND-gates and inverters. They can be used to represent and manipulate large Boolean functions in several applications such as equivalence checking and technology mapping. For many practical circuits, AIGs are smaller and easier to construct than BDDs. However, the applicability of traditional AIGs is limited because they are not canonical.

The paper presents a new way to compute functionally reduced AIGs (FRAIGs), *a variation of AIGs, which are "semi-canonical" in the sense that each AIG node has unique functionality among the nodes of the AIG. Previous methods perform functional reduction of traditional AIGs as a post-processing step. The new method performs functional reduction on-the-fly, as part of the AIG construction. This leads to significant savings in runtime by avoiding potentially large intermediate results. Preliminary experiments indicate an order-of-magnitude speed-up for typical applications.*

1 Introduction

AND-INV graphs (AIGs) are used to represent Boolean functions in combinational equivalence checking (CEC) [9][11], bounded model checking (BMC) [12][16], and technology mapping [13]. As a functional representation, AIGs enjoy several important properties:

- The construction time and the number of AIG nodes are proportional to the size of the original circuit (unlike BDDs, whose size is exponential for some important practical circuits, such as multipliers.)
- AIGs are composed of only two-input ANDs and inverters, represented as bubbled pointers (flipped bits) on the edges. This uniformity of representation gives some implementation advantages.
- AIGs coupled with Boolean satisfiability provide a powerful reasoning engine. The uniform structure of AIGs can be exploited by a circuit-based SAT solver, or translated into a CNF representation to be processed by a SAT solver.

However, AIGs [11] are not canonical; as a result, the same Boolean function can have many AIG representations. For example, function $F = abc$ can be represented as follows: $((ab)c)$, $(a(bc))$, $((ac)(bc))$, etc. Figure 1 shows two different AIGs of a four-variable function, which cannot be derived from each other by applying algebraic transformations. These AIGs are different Pareto points on the area/delay curve: one has fewer ANDs, while another has fewer levels of ANDs.

Because AIGs are not canonical, graphs constructed using traditional methods may have internal nodes with the same functionality. This may increase the number of AIG nodes and make reasoning on the AIG structure time consuming. Indeed, merging two functionally-equivalent nodes removes one variable from the SAT problem.

An AIG constructed by the traditional approach can be reduced using specialized algorithms [11][12]. However, proving functional equivalence of two AIG nodes may be a formidable task. Typically, it is solved with a SAT solver, which tries to prove that the outputs of the two AIGs never produce different values. In the published work, e.g. [15], detection of functional equivalence of AIG node-pairs (called *functional reduction* in this paper) is applied as a post-processing step.

The contribution of this paper is in integrating functional reduction into the traditional AIG construction. This leads to a semi-canonical data structure to represent Boolean functions, called *functionally reduced AIGs* (FRAIGs). The new construction algorithm is more robust in overcoming drawbacks of the traditional AIGs: large intermediate results and runtime overhead for the post-processing.

The algorithm proposed in this paper is similar to the efficient reduction-by-construction method [1] for Reduced Ordered Binary Decision Diagrams (ROBDDs), implemented in all the current ROBDD packages. Originally, the ROBDDs were introduced in [4] where the reduction process was applied as a post-processing step.

Experimental results confirm that the proposed method for constructing FRAIGs allows practical applications to run faster and to be applied to larger problem instances.

The paper is organized as follows. Section 2 surveys the traditional AIGs. Section 3 reviews previous work. Section 4 discusses the new algorithm to construct FRAIGs. Section 5 discusses some implementation details. Section 6 outlines some applications of FRAIGs. Section 7 reports experimental results. Section 8 concludes and outlines future work.

2 Background

This paper assumes familiarity with the basics of Boolean functions, Boolean networks, and Binary Decision Diagrams [2].

2.1 Definitions

Definition. *AND-INV graph* (AIG) is a Boolean network composed of two types of nodes: two-input AND-gates and inverters.

Definition. A representation of a Boolean function is *canonical* if, for any function, there exists only one representation of this type.

AIGs are not canonical, that is, the same function can be represented by two functionally equivalent AIGs, which have different structure. An example of such function is shown in Figure 1.

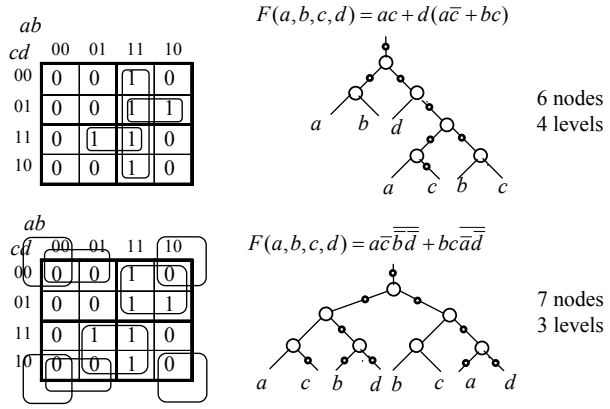

Figure 1. Two different AIGs for the same function.

Note that both graphs in Figure 1 are FRAIGs, since in each of them, no pair of nodes represent the same function.

Definition. The *size* of an AIG is the number of AND nodes in it. The number of logic levels is the number of AND-gates on the longest path from a primary input to a primary output.

The inverters are ignored when counting nodes and logic levels. In the software implementation, inverters are represented by flipping the least significant bit on the node pointers [11]. This implementation is similar to that of BDDs with complemented edges [1].

Definition. The function of an AIG node n, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in node n and expressed in terms of the PI variables x assigned to the leaf nodes of the AIG.

Definition. A *functionally reduced AIG* (FRAIG) is an AIG, in which, for any pair of nodes, n_1 and n_2, $f_{n_1}(x) \neq f_{n_2}(x)$ and $f_{n_1}(x) \neq \overline{f_{n_2}(x)}$.

2.2 AIG construction

AIGs for Boolean functions can be constructed starting from different functional descriptions:

SOP: Given an SOP representation of a function, AIGs of the products are constructed using the AIGs for elementary variables and cascades of two-input AND-gates. The AIG for the SOP output is constructed using the AIGs for the outputs of the product terms and a cascade of two-input OR-gates. Each two-input OR-gate is converted into a two-input AND-gate using the DeMorgan rule.

BDD: Given a (multi-output) BDD representation of a Boolean function, the (multi-output) AIG is constructed by converting the BDD into a circuit composed of MUXes and applying the transformation from the circuit representation.

Circuit: Given a circuit representation of a (multi-output) Boolean function, the (multi-output) AIG is constructed in a bottom-up fashion, by calling a recursive construction procedure for each PO of the circuit. The procedure checks if it is called for a PI node. If so, it returns the corresponding elementary AIG variable. Otherwise, it first calls itself for the fanins of a node and then builds the AIG for the node using the factored form or the BDD representation of the logic function of the node. In both cases, the elementary ANDs, ORs, and MUXes are converted into two-input ANDs and inverters.

Boolean formulas: Given an arbitrary expression representing a Boolean function using Boolean operators, including quantification and co-factoring, the AIG of the output of the formula is constructed as follows:

- Start with AIGs representing elementary variables.
- Express Boolean operations using two-input ANDs and inverters over the elementary variables.
- Perform co-factoring of a function by constructing the AIG of the function, followed by propagating corresponding constants through it.
- Perform existential (universal) quantification of a function by iteratively ORing (ANDing) the cofactors of the function w. r. t. the variables to be quantified.

When an AIG is constructed from a circuit, the number of AIG nodes does not exceed the number of literals in the factored forms of the nodes. When the AIG is constructed from a BDD, the number of AIG nodes does not exceed three times the number of nodes in the BDD. It follows that the size of the constructed AIG is proportional to the size of

the circuit or BDD. Quantifications performed on AIGs have the complexity exponential in the number of variables quantified. This is because quantifying each variable can potentially duplicate the graph size.

Boolean operations, except quantification, performed on AIGs lead to the resulting graphs, which, in the worst case, are not larger than the *sum* of the sizes of their arguments. Meanwhile, in the case of BDDs, the worst case complexity of the result is equal to the *product* of the sizes of the arguments. This difference explains why AIGs are more robust than BDDs for representing and manipulating complex circuitry, such as multipliers.

2.3 Structural hashing

Structural hashing (*strashing*) of AIGs introduces partial canonicity into the AIG structure. When a new AND-gate is added to the graph, several logic levels of the fanin AND-gates are mapped into a canonical form. Although the resulting AIG is not canonical, it contains sub-graphs, which are canonical as long as they have less than the given number of logic levels.

No strashing: When an AIG is constructed without strashing, AND-gates are added one at a time without checking whether an AND-gate with the same fanins already exists in the graph.

One-level strashing: When a new AND-gate is added, this type of strashing checks for a node with the same fanins (up to permutation).

Two-level strashing: This type has two phases. In the pre-computation phase, all two-level AND-INV combinations are enumerated and, for each Boolean function realizable by a two-level AIG, one representation is arbitrarily selected as the representative one. In the construction phase, when adding a new AND-gate, its two-level AIG is checked. If its canonical representative does not exist, the AND-gate is added. Otherwise, the canonical representation is constructed, even if it requires building new AND-gates for some of the fanins.

A detailed discussion of two-level structural hashing can be found in [8]. This reference uses an efficient implementation, which runs in time linear in the number of constructed AIG nodes. Its runtime is only marginally larger than that of one-level strashing, but the resulting graphs may have 5-20% fewer nodes.

A drawback of two-level strashing is when multiple AIGs are constructed repeatedly, sometimes it leads to an increase in the number of unused nodes in the AIG manager. This may slow down some AIG-based applications, such as image computation.

3 Previous work

AIGs have been applied as a circuit representation in combinational equivalence checking (CEC) [11] and an object graph representation in technology mapping [13]. In both cases, AIGs are built initially using strashing, and later optionally post-processed to enforce functional reduction.

In [15] AIGs are used for unbounded model checking in which both the circuits and interpolants computed from the unsatisfiability proofs are represented by AIGs. This work recognizes the need for functional reduction ([15], Section 3.2, paragraph 1) noting that AIGs tend to have many redundancies not captured by strashing.

Two procedures have been proposed to perform functional reduction. One, *bdd_sweep* [11], constructs BDDs of the AIG nodes in terms of the PI variables and intermediate "cut-point" variables. BDD construction is controlled by resource limits, such as a restriction on the BDD size. Any pair of AIG nodes with the same BDD is merged, and the fanout cones are rehashed to propagate the change. As long as all BDDs can be built within the resource limits, the result is a FRAIG.

A second procedure, *sat_sweep* [14][12], is more efficient and achieves the same merging and propagation by solving a sequence of incremental topologically-ordered SAT problems designed to prove or disprove the equivalence of cut-point pairs. Candidate pairs are detected by random simulation. Experimentally we will show that our on-the-fly FRAIGing method is an order of magnitude faster than this.

In both approaches, the initial graph is constructed in a redundant form, followed by functional reduction applied as a post-processing step.

Another approach to CEC was developed using NAND graphs [6] but the authors do not mention what methods are used to perform functional reduction or to prove the equivalence of the output functions.

4 Algorithm

This section presents the main contribution of the paper, a new and efficient algorithm to build AIGs on-the-fly while ensuring that they are functionally reduced by construction.

Figure 2 shows the pseudo-code of the traditional AIG construction with one-level strashing. The first part checks various trivial cases, such as when the nodes are equal up to complementation, or when one node is a constant. Next, the arguments are ordered to ensure that swapping of fanins does not create a new node.

One-level strashing is performed by looking up in a hash table, which maps the pair of fanins into the AND gate with these fanins. If a node with this pair of fanins exists, it is returned. If such node does not exist, a new node is created, added to the hash table, and returned.

```
Aig_Node * OperationAnd( Aig_Man * p, Aig_Node * n1, Aig_Nodee * n2 )
{
    Aig_Node * res, * cand, * temp;   Aig_NodeArray * class;

    /*** trivial cases ***/
    if ( n1 == n2 )                                    return n1;
    if ( n1 == NOT(n2) )                               return 0;
    if ( n1 == const )                                 return 0  or  n2;
    if ( n2 == const )                                 return 0  or  n1;
    if ( n1 < n2 )  { /*** swap the arguments ***/
        temp = n1; n1 = n2; n2 = temp;
    }
    /*** one level structural hashing ***/
    res = HashTableLookup( p->pTableStructure, n1, n2 );
    if ( res )                                         return res;
    res = CreateNode( p, n1, n2 );
    HashTableAdd( p->pTableStructure, res );           return res;
}
```

Figure 2. Algorithm for constructing AIGs with one-level strashing.

Figure 3 contains the pseudo-code of the FRAIG construction algorithm. Both one-level strashing and functional reduction are performed by the same procedure. An additional hash table is used, which maps each simulation vector into a set of functionally different AIG nodes that have this simulation vector (its *simulation class*).

```
Aig_Node * OperationAnd( Aig_Man * p, Aig_Node * n1, Aig_Nodee * n2 )
{
    Aig_Node * res, * cand, * temp;   Aig_NodeArray * class;
    /*** trivial cases ***/
    if ( n1 == n2 )                                    return n1;
    if ( n1 == NOT(n2) )                               return 0;
    if ( n1 == const )                                 return 0  or  n2;
    if ( n2 == const )                                 return 0  or  n1;
    if ( n1 < n2 )  { /*** swap the arguments ***/
        temp = n1; n1 = n2; n2 = temp;
    }
    /*** one level structural hashing ***/
    res = HashTableLookup( p->pTableStructure, n1, n2 );
    if ( res )                                         return res;
    res = CreateNode( p, n1, n2 );
    HashTableAdd( p->pTableStructure, res );
    if ( p->FlagUseOneLevelHashing )                   return res;

   /*** functional reduction ***/
    class = HashTableLookup( p->pTableSimulation, n1, n2 );
    if ( class == NULL ) {
        class = CreateNewSimulationClass( res );
        HashTableAdd( p-> pTableSimulation, class );   return res;
    }
    for each node cand in class
        if ( FunctionallyEquivalent( cand, res ) ) {
            AddNodeToEquivalenceClass( class, res );   return cand;
        }
    AddNodeToSimulationClass( class, res );            return res;
}
```

Figure 3. Algorithm for constructing FRAIGs.

The simulation vector is derived using bit-parallel simulation of the AIG starting from the PIs up to the node under construction. The simulation is performed incrementally whenever a new AND-gate is added. The simulation vector is derived by the bit-wise AND applied to (possibly complemented) simulation vectors of the fanins.

If the simulation class is empty, a new class is created and initialized with the given node. In this case, there is no need for the equivalence check because the new node is proved to be functionally unique by random simulation only.

If the simulation class is not empty, then for each representative, *cand,* of this class a SAT-based functional equivalence test is performed. Depending on the result of the test, two outcomes are possible. If the new node (*res*) is equivalent to the representative node (*cand*), then the representative node is returned to ensure functional reduction. The new node can be dropped. However, in the current implementation of FRAIGs, the new node is left in the graph as a node without fanouts. It is added to the equivalence class of the representative node as an alternative AIG structure. Finally, if the new node is not equivalent to any node in its simulation class, it is added to the simulation class and returned.

5 Implementation Details

5.1 Random Simulation

The performance of the proposed algorithm critically depends on the efficiency of random simulation. If the simulation vectors are larger, their distinguishing power is better, and fewer SAT-based functional equivalence tests are needed. In the current implementation, the default of 127 machine words is used to store bit-patterns at each node. Thus, roughly four thousand (127 * 32 = 4064) random bit-patterns are propagated through the circuit. The runtime of random simulation constitutes less than 5% of the total runtime, which is dominated by the SAT solver.

The default memory requirements for storing simulation information for one AIG node is 508 bytes (127 * 4 = 508), or approximately 5Mb per 10K of nodes. The memory used to store the simulation information is allocated independently from the memory used for the AIG nodes. Once the FRAIG construction is finished, the simulation memory can be de-allocated and re-used by the application.

5.2 SAT Solving

For efficiency, the algorithm requires tight integration of the circuit-based AIG data structure and a SAT solver. The solver used in the project is a state-of-the-art CNF-based solver MiniSat [7], with some minor modifications to restrict incremental SAT solving to a subset of variables and clauses.

The CNF for the AIG is loaded in the SAT solver incrementally, by adding three CNF clauses each time a new AIG node is created.

Checking functional equivalence for AIG nodes n_1 and n_2 is performed as follows: (1) collect the AIG nodes in the union of the transitive fanin cones of n_1 and n_2; (2) set the "branchable" variables to be those corresponding to the above AIG nodes; (3) run the solver to prove or disprove equivalence.

Incremental runs of the SAT solver create learned clauses, which are stored in the global clause database. Because the logic cones of different equivalence checking problems often overlap, the learned clauses are shared and reused, which improves the performance of the SAT solver.

6 Applications of FRAIGs

6.1 Formal Verification

In formal verification, FRAIGs can be used instead of the traditional AIGs as a data structure for CEC and BMC [9][11][12][16].

A straight-forward use of FRAIGs in CEC is similar to that of BDDs. FRAIGs are constructed for the circuit outputs. The circuits are equivalent if and only if the corresponding pairs of outputs are represented by the same FRAIG nodes.

A more sophisticated use of FRAIGs is to represent both circuits and interpolants in a uniform way, similar to [15]. This may extend the applicability of the previously reported model checking methods and lead to the development of new methods for sequential equivalence checking.

6.2 Logic Synthesis

A straight-forward use of FRAIGs in logic synthesis is to compact circuits by detecting and merging functionally equivalent nodes. Global FRAIGs for all the network nodes are constructed. Next, the network nodes are grouped into the same class if they are represented by the same FRAIG node. One representative of each class is selected and substituted for other nodes of the same class.

Other potential applications of FRAIGs in synthesis include: (a) a uniform representation of algebraic factored forms and DAGs resulting from Boolean decomposition, (b) a robust representation of node functions, manipulated by a logic synthesis system when it performs operations, such as elimination, collapsing, and node immunization, (c) an alternative computation engine to solve Boolean problems, such as don't-care computation.

6.3 Technology Mapping

A known approach to technology mapping [13] uses AIGs to represent the "object" graph. Of particular importance in this approach is implicit enumeration of mapping choices, achieved by collecting and storing multiple AIGs structures for the logic functions found in the original network to be mapped. If there are more mapping choices in the graph, the quality of mapping is better. In [13], mapping choices are derived by considering various algebraic decompositions of the SOPs at the nodes.

FRAIG construction can be seen as a natural way to prepare circuits for technology mapping. Each FRAIG node is associated with its equivalence class, that is, a set of functionally equivalent nodes with different AIG structures (structurally identical nodes are collapsed by one-level strashing performed as part of the FRAIG construction). These functionally equivalent nodes constitute a set of choices, which can be used for technology mapping.

An additional advantage is that FRAIGs can be constructed for multiple versions of the same network, derived by different optimizations. For example, a sequence of networks derived by applying an optimization script, one command at a time, can be "fraiged" into one object graph. Technology mapping applied to this cumulative graph selects the best mapping over all available choices, which may originate from different versions of the same network.

7 Experimental results

The proposed algorithm for constructing FRAIGs is implemented in C as a stand-alone AIG package "FRAIG" [17]. The package was tested in the MVSIS environment [18] and used in several applications dealing with logic synthesis and verification. Runtimes are reported on a 1.6GHz computer under Windows XP.

Several experiments were performed:

Experiment 1: Runtime comparison of synthesis operations in MVSIS using:

- FRAIGs
- Two-level strashing (MVSIS command *strash*).
- Strashing followed by incremental simulation-guided functional reduction applied to the AIG nodes in the topological order (MVSIS command *sat_sweep*).

Experiment 2: Runtime comparison during CEC in MVSIS using:

- FRAIGs
- BDD-based CEC (MVSIS command *verify*).
- Strashing-based CEC with monolithic SAT.
- Strashing-based CEC with incremental simulation-guided SAT (MVSIS command *sat_verify*).

7.1 Experiment 1

As a result of the first experiment it was found that, for the majority of MCNC [20] and ITC '99 [10] benchmarks, the runtime of FRAIG construction is only 2-3 times slower than that of two-level strashing. For larger benchmarks containing up to 20K gates, the runtime may be 10 times

slower. This difference is due to strashing complexity being linear in the size of the graph while FRAIG construction requires a linear number of SAT-based functional equivalence checks, each of which has a worst-case exponential complexity in the size of the graph. For larger benchmarks, the exponential behavior slows down the FRAIG construction.

The second part of this experiment compared the runtime of FRAIG construction with that of strashing followed by a post-processing step to enforce functional equivalence. For all the benchmarks tried, FRAIG construction was up to 10 times faster, because it avoids large redundant graphs appearing at the intermediate steps of construction.

A large combinational circuit *pj1.blif* extracted from the PicoJava benchmark [19] was selected for a case study. This circuit contains 17K gates and 35K literals after sweeping in MVSIS. It takes 0.3 sec to run *strash*, 31.0 sec to run *sat_sweep* and 2.7 sec to construct FRAIGs. The FRAIG runtime is divided as follows: 0.14 sec is spent on simulation, 0.24 sec for AIG traversal to detect the unions of TFI logic cones, and 1.86 sec for SAT solving.

7.2 Experiment 2

This experiment compares the performance of several CEC commands in MVSIS. CEC was used to prove functional equivalence of the original circuits against circuits derived using optimization scripts in MVSIS.

The original circuits are taken from the following sources:

- MCNC benchmarks [20] (the first four circuits)
- ISCAS benchmarks [3] (*s15850.blif*)
- PicoJava benchmarks [19] (*pj1.blif*)
- ITC'99 benchmarks [10] (*b14.blif, b17.blif*)

Most of these benchmarks were included in the tests because of their large sizes. Several smaller MCNC benchmarks were added to have circuits, for which BDDs could be constructed. The above selection of benchmarks used in the experiment is available on the web [17].

Table 1. Runtime comparison for CEC algorithms.

Name	Ins	Outs	Lits	BDD	SAT	SWEEP	FRAIG
des.blif	256	245	6084	0.3	1.0	2.8	0.5
c1355.blif	41	32	992	10.0	0.2	0.1	0.1
c6288.blif	32	32	4675	-	-	1.0	0.5
i10.blif	257	224	4355	57.2	2.4	1.5	0.3
s15850.blif	611	684	7303	6.3	1.5	3.3	0.5
pj1.blif	1769	1063	34533	-	-	31.9	10.5
b14.blif	32	54	17388	-	-	15.3	2.0
b17.blif	37	97	57311	-	-	385.6	13.2

The results are reported in Table 1. Column "Name" lists the benchmark name. Columns "Ins" ("Outs") show the number of PIs (POs). Column "Lits" is the number of literals in the factored forms after sweeping (removing single-input nodes and nodes without fanouts). The following four columns contain the runtimes in seconds, of the four algorithms. Memory needed to represent the circuits and solve the equivalence checking problem for the largest benchmark of the set, *b17.blif*, was 75Mb. The dash in Table 1 means that an algorithm could not complete after a timeout set to 600 seconds.

The CEC algorithms include verification by global BDD constriction (column "BDD"), verification by strashing followed by solving the resulting monolithic SAT problem (column "SAT"), verification by strashing followed by solving a sequence of incremental simulation-guided SAT problems (column "SWEEP"), and finally, verification through FRAIG construction (column "FRAIG") as described in Section 6.1.

8 Conclusions

Traditional AND-INV graphs (AIGs) [11] constructed with structural hashing [8] are not canonical because the construction algorithm does not guarantee that each node has a unique functionality. Practical applications rely on specialized procedures, such as *bdd_sweep* [11] or *sat_sweep* [14][12], to detect and eliminate functionally equivalent AIG nodes, which is important to control the AIG size and speed-up reasoning procedures.

This paper proposes an algorithm to build functionally reduced AIGs (FRAIGs), in which each node has a unique functionality by construction. The algorithm uses traditional structural hashing [8] as a quick pre-processing step, followed by random simulation [14][12] to detect a significant number of functionally unique nodes. Finally, when both methods fail, a local incremental SAT problem is solved to prove or disprove functional equivalence of the new node with the existing nodes.

Preliminary experiments include construction of FRAIGs for large benchmarks, and FRAIG-based CEC. The experiments confirm the usefulness of the proposed algorithm, which leads to an order-of-magnitude speed-up compared with known methods.

Future work will explore other potential applications of FRAIGs in logic synthesis, technology mapping, and equivalence checking, as outlined in Section 6.

References

[1] K. S. Brace, R. L. Rudell, R. E. Bryant, "Efficient implementation of a BDD package", *Proc. DAC '90*, pp. 40-45.

[2] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[3] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," Proc. IEEE Int'l Symp. on Circuits and Systems, 1989

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comp*., Vol. C-35, No. 8 (August, 1986), pp. 677-691.

[5] J. Cortadella, “Bi-decomposition and tree-height reduction for timing optimization”. *Proc. IWLS ’02*, pp. 233-238.

[6] R. Drechsler, M. Thornton, “Fast and efficient equivalence checking based on NAND-BDDs”, *Proc. VLSI ’01*.

[7] N. Eén, N. Sörensson, “An extensible SAT-solver”, *Proc. SAT ‘03*. http://www.cs.chalmers.se/~een/Satzoo/

[8] M. K. Ganai, A. Kuehlmann, “On-the-fly compression of logical circuits”. *Proc. IWLS ‘00*.

[9] E. Goldberg, M.Prasad, R.K.Brayton. “Using SAT for combinational equivalence checking”. *Proc. DATE ‘01*, pp. 114 -121.

[10] *ITC ’99 Benchmarks* http://www.cad.polito.it/tools/itc99.html

[11] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification”, *IEEE Trans. CAD*, Vol. 21(12), Dec 2002, pp. 1377-1394.

[12] A. Kuehlmann, “Dynamic Transition Relation Simplification for Bounded Property Checking”. *Proc. IWLS 2004*.

[13] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, “Logic decomposition during technology mapping,” *IEEE Trans. CAD*, 16(8), 1997, pp. 813-833.

[14] F. Lu, L. Wang, K. Cheng, R. Huang. “A circuit SAT solver with signal correlation guided learning”. *Proc. DATE ‘03*, pp. 892-897.

[15] K.L. McMillan, “Interpolation and SAT-based model checking”. *Proc. CAV ‘03*, pp. 1-13, LNCS 2725, Springer, 2003.

[16] K.L. McMillan, “Methods for exploiting SAT solvers in unbounded model checking”, *Proc. CAV 03.*

[17] A. Mishchenko. *FRAIG source code and benchmarks*. http://www.ee.pdx.edu/~alanmi/fraig

[18] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System.* UC Berkeley. http://www-cad.eecs.berkeley.edu/mvsis/

[19] SUN Microelectronics. *PicoJava Microprocessor Cores*. http://www.sun.com/microelectronics/picoJava/

[20] S. Yang. *Logic synthesis and optimization benchmarks*. Version 3.0. Tech. Report. Microelectronics Center of North Carolina, 1991.